Programy wielomodułowe

- w trakcie tworzenia dużych systemów oprogramowania celowe jest podzielenie systemu na mniejsze podsystemy, z których każdy jest oddzielnie analizowany i kodowany — umożliwia to racjonalne tworzenie i uruchamianie systemu;

- wygodnie jest umieścić kod realizujący funkcje takiego podsystemu w jednym lub kilku plikach, stanowiących razem zamkniętą całość, która może być poddana kompilacji — taki zespół plików (lub pojedynczy plik) nazywać będziemy *modułem programu*;

- w takim ujęciu cały system oprogramowania składa się z kilku modułów, które są oddzielnie kompilowane a potem integrowane (linkowane) w celu uzyskania programu wynikowego;

- poszczególne moduły mogą być pisane w tych samych bądź różnych językach programowania;

- kompilatory (i asemblery) dostępne w danym środowisku programistycznym generują kod wynikowy w tym samym języku, nazywanym *językiem pośrednim*;

- język pośredni można uważać za rodzaj "wspólnego mianownika" dla różnych kompilatorów; w środowisku komputerów PC kod w języku pośrednim (przechowywany w plikach z rozszerzeniem .OBJ) zawiera instrukcje procesora w formacie pozwalajacym na przeadresowywanie;

- proces integrowania kodu programu zapisanego w języku pośrednim nosi nazwę *linkowania* lub *konsolidacji*; w wyniku linkowania uzyskuje się plik wykonywalny (w komputerach PC plik z rozszerzeniem .EXE);

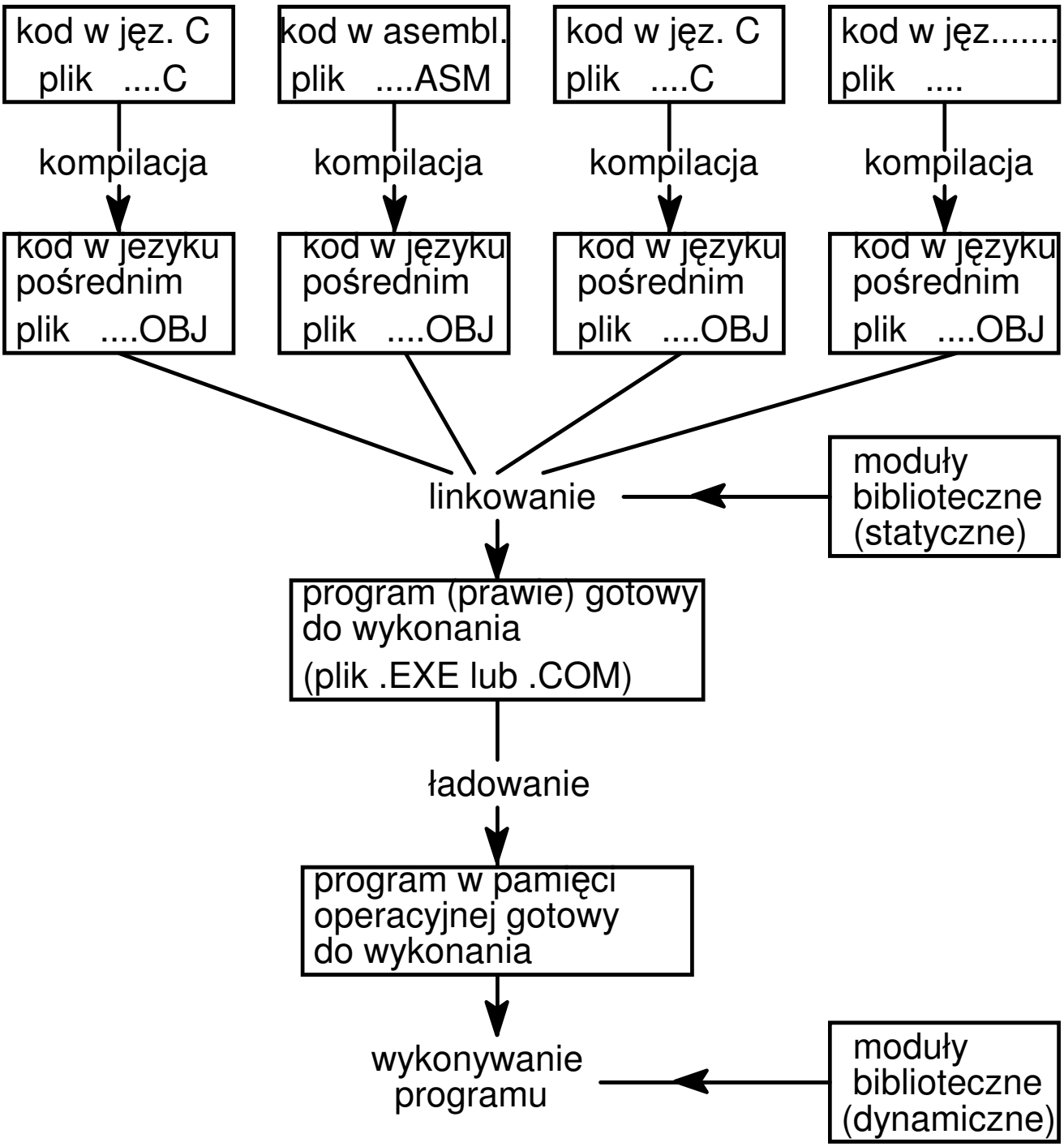
- zazwyczaj prócz plików zawierających skompilowany program, w procesie linkowania dołączane są też niezbędne podprogramy biblioteczne; zestawy takich podprogramów, zwane *bibliotekami statycznymi* przechowywane są zwykle w plikach z rozszerzeniem .LIB;

- niekiedy wygodniejsze może być udostępnienie podprogramów bibliotecznych dopiero w trakcie wykonywania programu — biblioteki takie nazywane są *dynamicznymi* i przechowywane są zazwyczaj w plikach z rozszerzeniem .DLL; zastąpienie bibliotek statycznych przez dynamiczne powoduje zmniejszenie rozmiaru pliku wykonywalnego .EXE;

- bezpośrednio przed rozpoczęciem wykonywania programu niektóre adresy instrukcji muszą być dostosowane do środowiska konkretnego komputera — korekcja adresów wykonywana jest w trakcie *ładowania* i korzysta z tablicy przeadresowań, która zapisana jest w pliku .EXE;

- po uruchomieniu systemu, od czasu do czasu wprowadza się drobne poprawki — zwykle wymagają one przeprowadzenia kompilacji jednego modułu, a następnie integracji (linkowania) wszystkich plików zawierających kod w języku pośrednim; linkowanie może dość czasochłonne, zwłaszcza przy dużej liczbie plików; z tego powodu niektóre środowiska programistyczne oferują mechanizm *linkowania inkrementalnego*;

- linkowanie inkrementalne polega na pozostawieniu pustego miejsca w pliku wykonywalnym, które przeznaczone jest na ewentualne późniejsze korekcje programu; w przypadku tworzenia poprawionej wersji programu nie przeprowadza ponownego pełnego linkowania, ale jedynie wprowadza poprawki do uzyskanego wcześniej pliku wykonywalnego (.EXE); po kilku kolejnych poprawkach pozostawione wolne miejsce wyczerpuje się i trzeba wykonać ponownie pełne linkowanie.


kod w jęz. C
plik C
kod w asembl.
plik ....ASM
kod w jęz. C
plik C
kod w jęz.....
plik ....
kompilacja
kompilacja
kompilacja
kompilacja
kod w języku pośrednim
plik ....OBJ
kod w języku pośrednim
plik ....OBJ
kod w języku pośrednim
plik ....OBJ
kod w języku pośrednim
plik ....OBJ
linkowanie
moduły biblioteczne (statyczne)
program (prawie) gotowy do wykonania
(plik .EXE lub .COM)
ładowanie
program w pamięci operacyjnej gotowy do wykonania
wykonywanie programu
moduły biblioteczne (dynamiczne)

Symbole lokalne i globalne

- w programach wielomodułowych powinny zostać określone ścisłe reguły dostępu do danych i przekazywania sterowania między poszczególnymi modułami;

- w wielu językach programowania obszar działania zmiennych i etykiet obejmuje tylko moduł, w którym zostały one zdefiniowane —w programie wielomodułowym trzeba więc rozszerzyć zasięg wybranych zmiennych i etykiet na cały program nadając im znaczenie globalne, tak by możliwy był dostęp do danych jak i przekazywanie sterowania między poszczególnymi modułami; zagadnienia te rozpatrzymy na przykładzie języka C i asemblera;

- program w języku C składa się z obiektów zewnętrznych, którymi mogą być *zmienne* i *funkcje*; zmienne mogą być definiowane na zewnątrz lub wewnątrz funkcji, natomiast funkcje są zawsze zewnętrzne (definiowanie funkcji wewnątrz innych funkcji jest niedozwolone);

- zmienne zadeklarowane wewnątrz funkcji (nazywane też *zmiennymi automatycznymi*) są dla niej lokalne, tj. żadna inna funkcja nie może mieć do nich bezpośredniego dostępu; takie zmienne tworzone są dopiero po wywołaniu funkcji i znikają po jej zakończeniu;

- zmienne lokalne wewnątrz funkcji mogą być deklarowane jako zmienne statyczne, np.

```
static int z ;
```

  zmienne wewnętrzne static, w odróżnieniu od zwykłych zmiennych wewnętrznych zachowują wartość między wywołaniami funkcji; innymi słowy zmienne wewnętrzne static stanowią prywatną pamięć funkcji;

- *zmienne zewnętrzne*, zwane też globalnymi są dostępne dla wszystkich funkcji; zmienne zewnętrzne definiuje się na zewnątrz wszystkich funkcji; zasięg zmiennej zewnętrznej i funkcji rozciąga się od miejsca, w którym została ona zadeklarowana w module (pliku) źródłowym do końca tego modułu (pliku);

- jeśli odwołanie do zmiennej zewnętrznej występuje przed jej definicją lub jeśli jej definicja znajduje się w innym module (w innym pliku źródłowym), to konieczne jest użycie deklaracji extern, np.

  Moduł I:
  ```
  int  p;                /* definicja zmiennej */
  ```

  Moduł II:
  ```
  extern  int  p;        /* deklaracja zmiennej */
  ```

  w module II można wykonywać operacje na zmiennej p dokładnie tak samo jak w module I (w którym zmienna została zdefiniowana);

- należy odróżniać definicję i deklarację zmiennej zewnętrznej:
  - definicja zmiennej zewnętrznej (np. int p ;) informuje o właściwościach zmiennej i powoduje rezerwację pewnej liczby bajtów pamięci, w których przechowywana będzie wartość zmiennej;
  - deklaracja zmiennej zewnętrznej (np. extern int p ;) informuje o właściwościach zmiennej;

- we wszystkich plikach składających się na jeden program źródłowy może wystąpić tylko jedna definicja każdej zmiennej zewnętrznej — pozostałe pliki mogą zawierać deklaracje extern zapewniające dostęp do tej zmiennej;

- zmienne zewnętrzne mogą być deklarowane jako zmienne statyczne, np.
  ```
  static  float  wsp_skali ;
  ```

- deklaracja static zastosowana do zmiennych zewnętrznych i funkcji ogranicza ich zasięg od miejsca wystąpienia do końca tłumaczonego pliku źródłowego (modułu); innymi słowy zmienna zewnętrzna staje się niewidzialna poza plikiem zawierającym jej deklarację;

- istnieje tendencja do szerokiego używania zmiennych zewnętrznych — w takim przypadku listy argumentów funkcji są krótkie, a zmienne zewnętrzne dostępne są bez ograniczeń; jednak w tego rodzaju programach powiązania między danymi nie są przejrzyste, przez co wszelkie modyfikacje programu stają się dość trudne i kłopotliwe;

- analogicznie deklaruje się funkcje, np.

```
extern  int  open_net (char *);
```

- deklarację static można stosować także do funkcji — jeśli nazwę funkcji zadeklarowano jako static, to nazwa funkcji jest niedostępna poza modułem (plikiem) zawierającym jej deklarację.

- w asemblerze obszar działania zmiennych i etykiet zasadniczo obejmuje tylko moduł, w którym zostały one zdefiniowane; dyrektywa PUBLIC nadaje zmiennej lub etykiecie zasięg globalny, obejmujący wszystkie moduły programu; dyrektywa EXTRN udostępnia zmienną lub etykietę globalną wewnątrz modułu; (omawiana tu dyrektywa PUBLIC nie ma nic wspólnego z dalej omawianym parametrem *public* dyrektywy SEGMENT);

- dyrektywa PUBLIC podaje listę nazw zmiennych, etykiet i symboli absolutnych (tj. symboli reprezentujących wartości stałe), np.

```
PUBLIC        okno, status, znak, wykonaj, param
```

- podane tu nazwy muszą być zdefiniowane w bieżącym module, np.

```
znak        EQU         'z'
param       EQU         0FFFEH
status      DW          ?
wykonaj     LABEL       near
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
okno        PROC        near
```

- na liście dyrektywy PUBLIC nie może wystąpić nazwa segmentu.

- nazwy globalne zdefiniowane w innych modułach (za pomocą dyrektywy PUBLIC) udostępnia dyrektywa EXTRN — ponieważ dyrektywa EXTRN w pewnym sensie zastępuje definicję zmiennej, etykiety czy symbolu absolutnego, więc obok każdej nazwy podaje się typ, np.

```
EXTRN       status : word,  okno : near
```

Rozwiązywanie odwołań zewnętrznych

- kod zawarty w plikach w języku pośrednim (.OBJ) zawiera również odwołania do danych i instrukcji znajdujących się w innych modułach; ponieważ jednak w trakcie kompilacji nie jest znana struktura programu wynikowego, odwołania te wyrażone są w postaci symbolicznej (np. pole adresowe rozkazu CALL pozostaje wyzerowane, ale zapisana jest dodatkowa informacja o nazwie wywoływanej funkcji);

- w trakcie linkowania, po wyznaczeniu położenia poszczególnych modułów w programie wynikowym, wyznacza się adresy w odwołaniach międzymodułowych — operacja ta określana jest często jako *rozwiązywanie odwołań zewnętrznych* (ang. resolve external references);

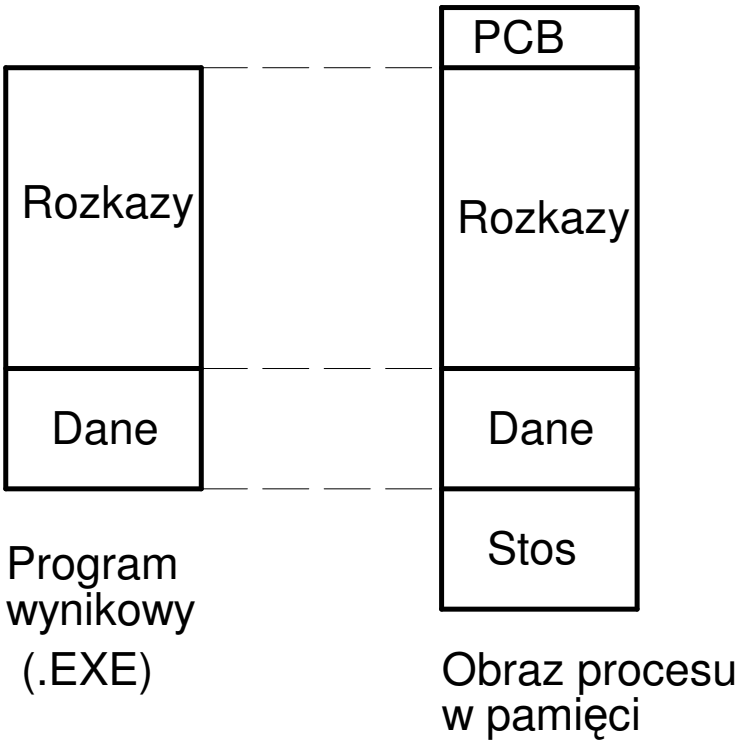
- rozwiązywanie odwołań zewnętrznych dotyczy także funkcji bibliotecznych włączanych do programu wynikowego w trakcie linkowania; jednak odwołania do funkcji bibliotecznych dołączanych dynamicznie (w trakcie wykonywania programu) pozostają nadal nierozwiązane — w takim przypadku rozwiązywanie odwołań wykonywane jest dopiero podczas ładowania lub wykonywania programu;

- w zależności od przyjętej techniki moduły biblioteczne mogą być ładowane do pamięci w trakcie ładowania programu, albo później, w trakcie wykonywania programu; w obu tych przypadkach po załadowaniu biblioteki do pamięci system operacyjny odpowiednio modyfikuje kod programu; adres funkcji bibliotecznej może być też przekazany do programu, co z kolei pozwala na wywołanie podprogramu za pomocą rozkazu skoku pośredniego;

- możliwość dynamicznego dołączania funkcji bibliotecznych jest charakterystyczna dla współczesnych systemów operacyjnych (np. Windows, Linux) i posiada sporo zalet:
  - pliki zawierające program wynikowy (np. EXE) są krótsze;
  - kody funkcji bibliotecznych mogą być poprawiane i ulepszane, bez konieczności ponownego linkowania programu;
  - kod rozkazowy funkcji bibliotecznych może być współdzielony przez kilka procesów;
  - biblioteki mogą być opracowywane i udostępniane przez niezależnych wytwórców;
  - w pewnych zastosowaniach (np. transakcje w bazach danych) wybór modułu bibliotecznego następuje dopiero w trakcie wykonywania programu, w zależności od życzeń użytkownika; znane są przypadki, w których nazwa funkcji bibliotecznej jest zależna od konfiguracji komputera (nazwa podana jest w pliku konfiguracyjnym).

Ładowanie programów

- w klasycznym ujęciu ładowanie programu do pamięci polega na przydzieleniu odpowiednio dużego obszaru pamięci, przeznaczonego na blok PCB (ang. process control block), rozkazy, dane i stos, a następnie przepisaniu kodu danych i danych programu z pliku dyskowego do pamięci operacyjnej;



- proces ładowania we współczesnych komputerach jest zazwyczaj bardziej skomplikowany;

- w trakcie linkowania programu nie jest znane położenie programu w pamięci w trakcie jego wykonywania; często przyjmuje się, że program zostanie umieszczony w pamięci począwszy od komórki o adresie 0;

- zazwyczaj nie jest możliwe umieszczenie programu począwszy od komorki 0, jednak pewne architektury procesorów (np. 8086/88) pozwalają na umieszczenie programu w dowolnym obszarze pamięci i poprawne wykonanie go, nawet jeśli program został linkowany przy założeniu, że zostanie umieszczony począwszy od adresu 0;

- w przypadku procesora 8086/88 dotyczy to jednak tylko programów, których rozmiar jest mniejszy od 64KB; programy tego typu, zapisywane w plikom z rozszerzeniem .COM (zob. dalszy opis), wykonywane są przy stałych zawartościach rejestrów segmentowych CS=DS=SS, które ustawiane są przez system operacyjny przed uruchomieniem programu; ponieważ adres fizyczne rozkazów i danych obliczane są wg tego samego schematu (rej. segmentowy * 16 + offset), więc kod i dane programu mogą być umieszczone w dowolnym obszarze pamięci, którego początek wskazują rejestry CS, DS i ES;

- niezależnie od tego adresy rozkazów sterujących (skoków) w omawianym procesorze obliczane są względem bieżącej zawartości wskaźnika instrukcji (E)IP wg formuły *(E)IP + zaw. pola adresowego*; jeśli więc pewien fragment programu zostanie przesunięty w inne miejsce, to rozkaz skoku będzie nadal omijał (przeskakiwał) ustaloną liczbę rozkazów;

- wyłoniły się trzy podstawowe techniki ładowania programów do pamięci:
  - ładowanie bezwzględne,
  - ładowanie relokowalne,
  - ładowanie dynamiczne;

- w przypadku *ładowania bezwzględnego* cały kod programu musi być w pełni określony w chwili ładowania go do pamięci; programista może posługiwać adresami fizycznymi lokacji pamięci, lub też adresy te mogą wyznaczone przez kompilator (asembler) i linker; niezbędna jest jednak znajomość adresu początkowego obszaru pamięci, do którego zostanie wpisany program;

- znacznie bardziej elastyczne są rozwiązania, w których decyzję o położeniu programu w pamięci podejmuje się podczas ładowania — mówimy wówczas o *ładowaniu relokowalnym*; w takim przypadku program można umieścić w dowolnym obszarze pamięci; technika ładowania relokowalnego stosowana jest m.in. w odniesieniu do plików w formacie DOS EXE (zob. dalszy opis);

- *ładowanie dynamiczne* jest charakterystyczne dla współczesnych wielozadaniowych systemów operacyjnych, w których stosuje się pamięć wirtualną; wówczas poszczególne fragmenty programu mogą być rozproszone w pamięci operacyjnej, a część z nich może być przechowywana na dysku; ten sam fragment programu, jeśli zostanie ponownie przepisany z dysku do pamięci operacyjnej, może pracować w obszarze o innych adresach fizycznych; realizacja ładowania dynamicznego wymaga więc skomplikowanych sprzętowych mechanizmów transformacji adresów (zob. stronicowanie).

Pliki linkowalne i wykonywalne

- pliki zawierające kod i dane programu, zakodowane w sposób zrozumiały dla procesora, przyjęto określać terminem angielskim *object file*; pliki takie zawierają dodatkowe informacje opisujące strukturę i własności programu, a także inne dane potrzebne do uruchomienia programu;

- plik *object* może być:
  - *linkowalny* (ang. linkable), czyli może być przygotowany do przetwarzania przez konsolidator (linker); plik linkowalny obok właściwego kodu zawiera obszerne informacje o używanych symbolach i wymaganiach relokacji; niekiedy kod podzielony jest na segmenty logiczne, które mogą indywidualnie przetwarzane przez linker;
  - *wykonywalny* (ang. executable), czyli może być załadowany do pamięci i wykonywany jako program; plik zawiera kod i dane, ale nie zawiera informacji o symbolach (chyba że przewiduje się linkowanie dynamiczne); nie występują także informacje o relokacji (ewent. bardzo ograniczone); struktura pliku wykonywalnego stanowi odbicie specyfiki środowiska, w którym wykonywany jest program;
  - *ładowalny* (ang. loadable) czyli może być załadowany do pamięci jako biblioteka razem z programem;
  - mogą też występować kombinacje trzech ww. możliwości;

- w systemie Windows/DOS pliki linkowalne są kodowane w formacie OMF, a wykonywalne w formacie EXE — oba te formaty są mają odmienną budowę;

- w wymienionych typach plików można wyróżnić pięć rodzajów informacji (chociaż nie każdy plik zawiera te informacje):
  - *nagłówek* (ang. header) — zawiera ogólne informacje o pliku, data tworzenia, rozmiar kodu, itp.;
  - *kod wykonywalny* (ang. object code) — instrukcje i dane w postaci binarnej, wygenerowane przez kompilator lub asembler;
  - *opis relokacji* — lista miejsc w kodzie wykonywalnym, które muszą być skorygowane;
  - *symbole* — symbole globalne zdefiniowane w danym module, symbole importowane z innych modułów lub symbole definiowane przez linker;
  - *informacje dla debuggera* — dodatkowe informacje potrzebne dla debuggera, jak np. symbole lokalne, numery linii programu źródłowego, opisy struktur danych, itp.

Format COM

- format COM stosowany w systemie DOS stanowi skrajny przykład pliku, zawierającego wyłącznie kod binarny;

- bezpośrednio przed rozpoczęciem wykonywania programu system operacyjny ładuje plik do wolnego obszaru pamięci; początkowe 256 bajtów tego obszaru zajmuje blok PCB, tu nazywany *przedrostkiem segmentu programu* (ang. PSP — program segment prefix);

- zawartość pliku wpisywana jest do obszaru począwszy od offsetu 100H; rejestry segmentowe są ustawiane tak, że wskazują początek bloku PSP, rejestr SP wskazuje koniec segmentu, po czym następuje skok do początku programu;

- możliwość zdefiniowania takiego prostego formatu wynika z cech architektury segmentowej procesora 8086/88 — ponieważ adresy są obliczane względem początku segmentu, a początek segmentu wskazywany jest przez rejestr segmentowy, więc niezależnie od zajętości pamięci program może zostać załadowany od offsetu 100H; zatem dla programów jednosegmentowych nie są potrzebne żadne korekcje adresów, ponieważ adresy względem początku segmentu mogą być obliczone w trakcie linkowania;

- jeśli program nie mieści się w jednym segmencie, to zachodzi konieczność dostosowania niektórych adresów do rzeczywistego położenia programu w pamięci — spotykane są wyjątkowo programy w formacie COM, które wykonują to samodzielnie.

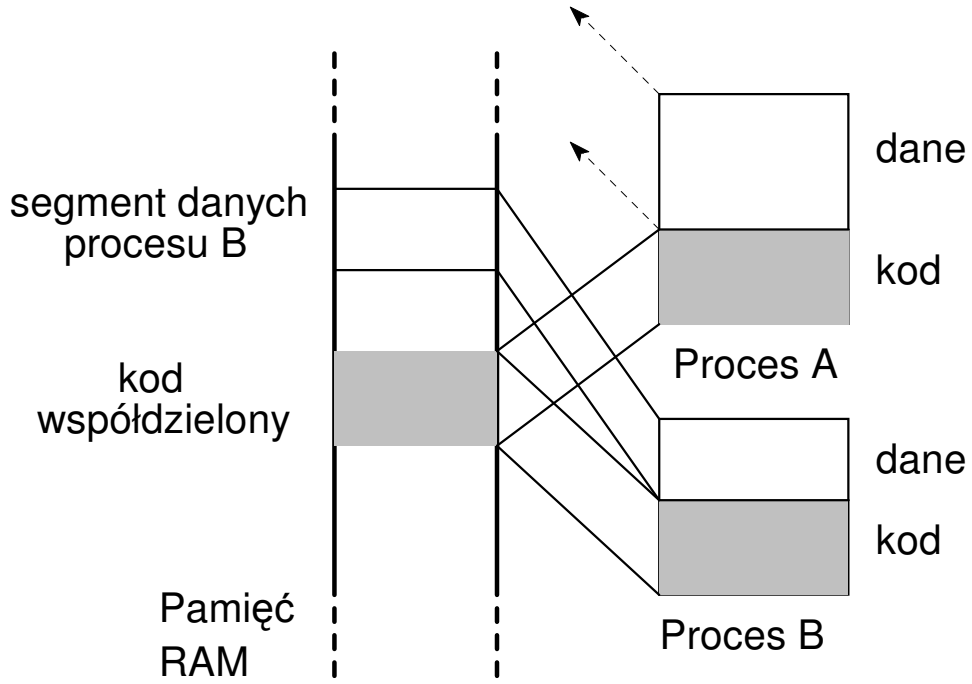
Format a.out w systemie Unix

- prawie wszystkie używane obecnie komputery posiadają możliwość sprzętowej relokacji pamięci; w takich komputerach nowy proces tworzony jest w pustej przestrzeni adresowej, a więc program może być linkowany przy założeniu, że zostanie zapisany do ustalonego obszaru pamięci — zatem nie jest potrzebna relokacja w trakcie ładowania; format znany jako a.out w systemie Unix dotyczy omawianej sytuacji;

- w najprostszym przypadku plik a.out zawiera mały nagłówek, za którym umieszczony jest kod wykonywalny (z powodów historycznych nazywany *text*) i wartości początkowe danych statycznych;

- wiele wczesnych instalacji systemu Unix zostało używało procesora PDP-11, którego architektura wywarła znaczny wpływ na procesory konstruowane w latach siedemdzisiątych i osiemdziesiątych; PDP-11 stosuje tylko adresowanie 16-bitowe, co ogranicza rozmiar programu do 64 KB; w późniejszych wersjach PDP-11 wprowadzono oddzielną przestrzeń adresową dla danych (D) i oddzielną dla kodu (I), wskutek czego program mógł zawierać 64 KB kodu i 64 KB danych; zatem programy tworzone dla PDP-11 musiały zawierać dwie sekcje, a program ładujący umieszczał pierwszą sekcję w przestrzeni adresowej I, a drugą sekcję w przestrzeni D;

- dodatkowa zaleta: można było uruchamiać kilka kopii programu, które korzystały z tego samego kodu, chociaż każda z nich miała własny segment danych; w systemach z podziałem czasu (np. Unix) pracuje wiele kopii interpretera zleceń (ang. shell) i demonów sieciowych, co powoduje znaczną oszczędność miejsca w pamięci;



- we współczesnych komputerach z dużą przestrzenią adresową na ogół istnieje możliwość definiowania stron w pamięci wirtualnej "tylko do odczytu" i stron "odczyt/zapis" — zatem w stosowanych formatach powinna istnieć możliwość zaznaczania sekcji "tylko do odczytu" i sekcji "odczyt/zapis";

- nagłówek pliku w formacie a.out zmienia się w różnych wersjach Unixa, ale podana niżej struktura nagłówka stosowana w wersji Unixa BSD jest typowa; w strukturze tej przyjęto, że wartości int są 32-bitowe, a short int 16-bitowe.

```
int   a_magic;      // magic number
int   a_text;       // rozmiar segmentu kodu (text segment)
int   a_data;       // rozmiar segmentu danych inicjalizowanych
                    // (initialized data)
int   a_bss;        // rozmiar segmentu danych nieinicjalizowanych
                    // (uninitialized data)
int   a_syms;       // rozmiar tablicy symboli (symbol table)
int   a_entry;      // punkt wejścia
int   a_trsize;     // rozmiar relokacji kodu (text relocation size)
int   a_drsize;     // rozmiar tablicy relokacji danych
                    // (data relocation size)
```

- pole a_magic określa rodzaj pliku wykonywalnego (executable); historycznie: "magic number" w oryginalnych PDP-11 miała wartość oktalnie 407, co było równoważne instrukcji skoku, która przeskakiwała 7 kolejnych słów nagłówka (header) do początku segmentu kodu (*text segment*); stanowi to pewną prymitywną formę kodu niezależnego od położenia: program ładujący (ang. boostrap loader) ładuje całą pamięć, zwykle od lokacji 0 i następnie skacze do początku załadowanego pliku; tylko kilka programów korzysta z tego, ale liczba 407 przetrwała 25 lat;

- pola a_text i a_data określają rozmiary segmentu kodu (tylko odczyt) i danych (odczyt/zapis) w bajtach;

- Unix automatycznie zeruje każdy przydzielany obszar pamięci, więc dane o wartościach początkowych 0 lub dane, których wartość początkowa jest nieistotna, nie są umieszczane w pliku a.out; rozmiar obszaru niezainicjalizowanego (ściśle: zainicjalizowanego zerami) określa pole a_bss;

- pole a_entry podaje adres punktu wejścia do programu;

- pola a_syms, a_trsize i a_drsize informują o wielkości tablicy symboli i informacji o relokacji, które podane są w pliku za segmentem danych; pola te zawierają zera w programach zlinkowanych (chyba że są podane symbole dla debuggera);

- proces ładowania i uruchamiania pliku 2-segmentowego przebiega następująco:
  - odczyt nagłówka a.out w celu uzyskania rozmiaru segmentów;
  - sprawdzenie, czy istnieje już współdzielony segment kodu (ang. sharable code segment) dla tego pliku; jeśli tak, to segment odwzorowywany jest w przestrzeń adresową procesu; jeśli nie, to tworzy się obszar i czyta się segment kodu z pliku do tego obszaru;
  - tworzenie prywatnego segmentu danych odpowiednio dużego dla danych i BSS;
  - tworzenie i odwzorowanie segmentu stosu (zwykle oddzielnie od segmentu danych, ponieważ sterta i stos rosną niezależnie); umieszczenie parametrów linii zlecenia na stosie;
  - odpowiednie ustawienie rejestrów i skok do początku programu;

- dostęp do danych poszczególnych procesów (zawartych w odrębnych segmentach), i ewentualne współdzielenie segmentu kodu może być zrealizowane za pomocą adresowania wykorzystującego rejestry bazowe (segmentowe);

- podany schemat (znany jako NMAGIC = New MAGIC) używany był od r. 1975 na komputerach PDP-11 i VAX; gdy pojawiła się pamięć wirtualna, wprowadzono kilka ulepszeń, które przyśpieszały ładowanie i oszczędzały pamięć;

- w systemach ze stronicowaniem, podany powyżej prosty schemat przydziela pamięć wirtualną dla każdego segmentu kodu i dla każdego segmentu danych; ponieważ plik a.out jest już przechowywany na dysku, więc może być traktowany tak jak gdyby stanowił część pliku wymiany; rozwiązanie to pozwala zaoszczędzić miejsce na dysku, a jednocześnie uruchamianie programu trwa krócej ponieważ wystarczy załadować z dysku tylko te strony, które program aktualnie używa, a nie cały plik;

- w wyniku wprowadzenia kilku zmian utworzono format znany jako ZMAGIC; zmiany dotyczą dopasowania (ang. align) segmentów w pliku do granic stron; w systemie ze stronami 4 kB, nagłówek pliku a.out jest rozszerzany do 4 kB, a rozmiar segmentu kodu jest zaokrąglany w górę do wielokrotności 4 kB;

- pliki ZMAGIC redukują zbędne stronicowanie, ale ceną jest niewykorzystanie dużej przestrzeni dyskowej — nagłówek pliku a.out ma tylko 32 bajty, a zajmuje 4 kB; średnio traci się 2 kB na granicy segmentu kodu i segmentu danych; obie te wady są usunięte w formacie QMAGIC;

- Unix BSD ładuje programy od lokacji 0 (lub od 0x1000 dla QMAGIC); inne wersje Unixa ładują programy od innych adresów, np. System V dla Motoroli 68K ładuje od 0x80000000, a dla 386 ładuje od 0x8048000.

Format DOS EXE

- format a.out jest odpowiedni dla systemów, w których dla każdego procesu przydzielana jest odrębna przestrzeń adresowa, tak że każdy program może być ładowany od tego samego adresu logicznego; 32-bitowe wersje Windows ładują wprawdzie programy do odrębnej przestrzeni adresowej, ale adresy początkowe obszarów nie zawsze są jednakowe; podobnie w systemie DOS początkowy adres programu jest ustalany dopiero bezpośrednio przed załadowaniem programu;

- w takich przypadkach plik wykonywalny musi zawierać zestawienie miejsc w programie, w których znajdujące się tam adresy muszą być zmodyfikowane w trakcie ładowania programu; takie zestawienie w literaturze angielskiej nazywane jest *relocation entries* lub *fixups*; format DOS EXE zaliczany jest do stosunkowo prostych formatów, w których używana jest tablica relokacji;

- każdy plik w formacie EXE rozpoczyna się od nagłówka o podanej niżej strukturze

```
char signature[2] = "MZ"; // identyfikator formatu (magic number)
short lastsize;     // długość pliku mod 512
short nblocks;      // liczba bloków 512 zajmowanych przez plik
short nreloc;       // liczba pozycji w tablicy modyfikacji obrazu
                    // programu
short hdrsize;      // rozmiar nagłówkach w (16-bajtowych)
                    // paragrafach
short minalloc;     // minimalna liczba przydzielanych
                    // dodatkowych paragrafów
short maxalloc;     // maksymalna liczba przydzielanych
                    // dodatkowych paragrafów
void far * sp;      // zawartość początkowa rejestrów SS:SP
short checksum;     // suma kontrolna
void far * ip;      // zawartość początkowa rejestrów CS:IP
short relocpos;     // położenie tablicy modyfikacji obrazu
                    // programu (ang. location of relocation
                    // fixup table)
short noverlay;     // liczba nakładek programu
char extra[ ];      // dodatkowy obszar dla nakładek, itp.
void far * relocs[ ];// kolejne pozycje tablicy relokacji
```

- na początku pliku umieszczone są znaki MZ identyfikujące plik EXE;

- dwa kolejne pola (lastsize, nblocks) określają rozmiar pliku, np. jeśli:

  ilość stron 512-bajtowych = 3,
  długość pliku mod 512 = 1

  to plik zawiera 1025 bajtów;

- rozmiar nagłówka wyrażony w paragrafach (jednostkach 16-bajtowych) zawiera pole hdrsize o adresie względnym 08H;

- dane niezanicjalizowane umieszczone na końcu programu (np. segment stosu) nie są włączane do obrazu programu, co zmniejsza rozmiary pliku EXE — wielkość obszaru pamięci zajmowanego przez te dane reprezentuje pole minalloc; wartość podana w tym polu wyrażona jest w paragrafach (jednostkach 16-bajtowych);

- warunkiem załadowania programu do pamięci operacyjnej jest, by system DOS dysponował niezajętym ciągłym obszarem pamięci zawierającym co najmniej:

  (*rozmiar PSP*) + (*rozmiar obrazu programu*) + minalloc [bajtów]

- pole maxalloc zawiera wielkość dodatkowego obszaru pamięci, który może ułatwiać pracę programu, nie jest jednak niezbędny do jego wykonywania;

- zawartość pola maxalloc określona jest przez parametr linkowania /CPARMAXALLOC: (w skrócie /C:), np. /C:20 ; wartość podana w polu maxalloc wyrażona jest także w paragrafach;

- pominięcie parametru linkowania /CPARMAXALLOC powoduje wpisanie wartości standardowej FFFFH, co praktycznie oznacza że program zajmie największy dostępny obszar pamięci, którym dysponuje system DOS; może to być źródłem pewnych trudności jeśli program próbuje wykonywać inny program jako proces potomny;

- jeśli maxalloc = minalloc = 0, to program ładowany do górnego obszaru pamięci (poniżej A0000H);

- pliki EXE nowszych typów można rozpoznać poprzez analizę adresu 4-bajtowego umieszczonego pod adresem względnym 3CH — jeśli adres ten wskazuje na znaki NE, LE, PE, itd., to plik EXE posiada strukturę nowego typu; wewnątrz takiego pliku znajduje się zazwyczaj także krótki program przystosowany do pracy w systemie DOS (ang. stub), którego wykonanie powoduje wyświetlenie krótkiego komunikatu, np. "This program must be run under Microsoft Windows"; format PE opisany jest dalej;

- analizę programów w różnych formatach EXE można przeprowadzać za pomocą różnych programów analizujących — najczęściej spotykane są:
  EXEHDR (Microsoft),
  TDUMP (Borland).

- ładowanie do pamięci programu w formacie EXE przebiega następująco:
  - odczytanie nagłówka, sprawdzenie identyfikatora;
  - przydzielenie odpowiedniego obszaru pamięci — pola minalloc i maxalloc określają minimalną i maksymalną liczbę dodatkowych paragrafów pamięci, które mają być przydzielane na końcu załadowanego programu;
  - tworzenie bloku PSP;
  - czytanie kodu programu (rozmiar określony jest przez pola nblocks i lastsize;
  - wykonanie modyfikacji obrazu programu wg tablicy relokacji;
  - skorygowanie początkowych zawartości rejestrów SS i CS.

Modyfikacja obrazu programu w formacie DOS EXE

- program konsolidujący (linker) tworzy program przy założeniu, że zostanie on umieszczony w pamięci począwszy od komórki o adresie 0000:0000 — zazwyczaj nie jest to możliwe ponieważ obszar początkowy zajmują różne zmienne i tablice systemowe oraz sam system operacyjny; z tego względu bezpośrednio przed rozpoczęciem wykonywania programu konieczna jest korekcja niektórych adresów podanych w programie;

- korekcja konieczna jest także w odniesieniu do początkowych zawartości rejestrów CS i SS przechowywanych w nagłówku; jeśli program zostanie umieszczony w pamięci począwszy od adresu segmentowego *k*, to do rejestru CS zostanie wpisana suma zawartości pola CS i liczby *k*; analogicznie SS; pola IP oraz SP przepisywane są do odpowiednich rejestrów bez zmian;

- korekcja adresów zawartych w programie dotyczy przede wszystkim instrukcji, których operandy zostały określone za pomocą operatora SEG albo też pole adresowe zawiera odwołanie do innego segmentu (instrukcje CALL FAR, JMP FAR);

- każda pozycja w tablicy modyfikacji obrazu programu zawiera 4-bajtowy adres w układzie *segment:offset* — adres ten wskazuje położenie słowa (16-bitowego), które powinno zostać zwiększone o adres (segmentowy) pierwszego segmentu programu;

- liczba pozycji w tablicy modyfikacji obrazu programu umieszczona jest nagłówku pliku (pole nreloc).

Przykład — poniżej podano przykładowy program w asemblerze oraz fragmenty pliku EXE utworzonego w rezultacie jego translacji.

```
rozkazy  SEGMENT
         assume  cs:rozkazy
pocz:    nop
         mov     ah, 09H             ; wyświetlanie łańcucha
         mov     dx, SEG znaki       ; znaków
         mov     ds, dx
         mov     dx, OFFSET znaki
         int     21H
         mov     ax, 4C00h
         int     21H
znaki    db      'Testowanie .EXE', 0DH, 0AH, '$'
rozkazy  ENDS
stos_pr  SEGMENT   stack
         dw      320 dup (?)
stos_pr  ENDS
         END     pocz
```

```
0000H   4D  5A  31  00  02  00  01  00
        20  00  29  00  29  00  04  00
0010H   80  02  50  8C  00  00  00  00
        1E  00  00  00  01  00  04  00
0020H   00  00  00  00  00  00  00  00
        00  00  00  00  00  00  00  00
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
01F0H   00  00  00  00  00  00  00  00
        00  00  00  00  00  00  00  00
0200H   90  B4  09  BA  00  00  8E  DA
        BA  11  00  CD  21  B4  4C  CD
0210H   21  54  65  73  74  6F  77  61
        6E  69  65  20  70  72  6F  67
0220H   72  61  6D  75  20  74  79  70
```

```
          75   20   2E   45   58   45   0D   0A
0230H     24
```

Format ELF w systemie Unix

- format a.out używany był w Unixie ponad dziesięć lat, ale wraz z pojawieniem się wersji *System V* zauważono, że potrzebne są nowe własności do wspomagania kompilacji skrośnej (ang. cross-compilation), linkowania dynamicznego i innych cech współczesnych systemów;

- wczesne wersje systemu V używały formatu COFF (ang. Common Object File Format), który pierwotnie został zaprojektowany dla kompilacji skrośnej w systemach wbudowanych i nie pracował dobrze w systemach z podziałem czasu — bez rozszerzeń nie mógł obsługiwać C++ lub linkowania dynamicznego;

- w poźniejszych wersjach systemu V, format COFF został zastąpiony przez format ELF (ang. Executable and Linking Format); ELF został przyjęty w systemie Linux, jak również w odmianach Unixa BSD; znane są 32- i 64-bitowe wersje formatu ELF;

- pliki ELF występują w trzech odmianach: linkowalny, wykonywalny i "shared object":
  - pliki linkowalne tworzone są przez kompilatory i asemblery, ale przed wykonaniem muszą być przetworzone przez linker;
  - pliki wykonywalne (ang. executable) mają już zrealizowane wszystkie relokacje i wszystkie symbole rozwiązane (ang. resolved), z wyjątkiem symboli stanowiących odwołania do współdzielonych bibliotek, które będą rozwiązane w trakcie wykonywania programu;
  - "shared objects" są współdzielonymi bibliotekami, zawierającymi zarówno informację symboliczną dla linkera i bezpośrednio wykonywalny kod dla czasu wykonywania;

- pliki ELF zaczynają się od nagłówka ELF; nagłówek jest zaprojektowany do dekodowania nawet w komputerach z innym porządkiem bajtów niż w architekturze docelowej dla pliku; pierwsze cztery bajty stanowią identyfikator ("magic number"), który identyfikuje plik ELF, po których następują trzy bajty opisujące format reszty nagłówka; po przeczytaniu znaczników class i byteorder program zna porządek bajtów i rozmiar słowa — można wówczas wykonać konieczną zamianę bajtów; inne pola, jeśli są podane, podają rozmiar i położenie nagłówka sekcji i programu;

```
char magic[4] = "\177ELF";      // identyfikator formatu
                                // (magic number)
char class;         // rozmiar adresów: 1 = 32 bity, 2 = 64 bity
char byteorder;     // porządek bajtów: 1 = mniejsze niżej (little-
                    // endian), 2 = mniejsze wyżej (big-endian)
char hversion;      // wersja nagłówka (zawsze 1)
char pad[9];
short filetype;     // typ pliku: 1 = linkowalny (relokowalny),
                    // 2 = wykonywalny, 3 = shared object,
                    // 4 = obraz pamięci (core image)
short archtype;     // typ procesora: 2 = SPARC, 3 = x86,
                    // 4 = 68K, etc.
int fversion;       // wersja pliku (zawsze 1)
int entry;          // punkt wejścia (entry point) jeśli plik
                    // wykonywalny
int phdrpos;        // file position of program header or 0
int shdrpos;        // file position of section header or 0
int flags;          // architecture specific flags, usually 0
short hdrsize;      // rozmiar nagłówka pliku ELF
short phdrent;      // size of an entry in program header
short phdrcnt;      // number of entries in program header or 0
short shdrent;      // size of an entry in section header
short phdrcnt;      // number of entries in section header or 0
short strsec;       // section number that contains section name
                    // strings
```

Przykładowy nagłówek pliku w formacie ELF
(komputer "juggernaut" z procesorem Sparc)

7F 45 4C 46	identyfikator formatu (ELF)
01	rozmiar adresów (tu: 32-bitowe)
02	mniejsze wyżej
01	wersja nagłówka (zawsze 1)
00 00 00 00 00 00 00 00 00	(char pad[9];)
00 02	typ 2 = wykonywalny
00 02	procesor SPARC
00 00 00 01	wersja pliku (zawsze 1)
00 01 08 28	punkt wejścia
00 00 00 34	(int phdrpos;)
00 00 80 10	(int shdrpos;)
00 00 00 00	(int flags;) zwykle 0
00 34	rozmiar nagłówka ELF
00 20	(short phdrent;)
00 05	(short phdrcnt;)
00 28	(short shdrent;)
00 1D	(short phdrcnt;)
00 1A	(short strsec;)

Własności formatu ELF

- ELF jest umiarkowanie złożonym formatem, który służy wielu celom: z jednej strony jako format relokowalny może być używany dla C++, z drugiej strony stanowi efektywny format wykonywalny dla systemów z pamięcią wirtualną i linkowaniem dynamicznym; jednocześnie pozwala na łatwe odwzorowanie stron kodu wykonywalnego na przestrzeń adresową programu;

- umożliwia także kompilację skrośną z jednej platformy na inną — plik ELF zawiera odpowiednie informacje potrzebne do identyfikacji typu procesora i kolejności bajtów.

Format PE (Portable Executable)

- w konstrukcji systemu Windows NT można odnaleźć wiele wpływów innych systemów operacyjnych (DOS, wczesne wersje Windows, system VAX VMS firmy Digital, Unix System V);

- format przyjęty w systemie NT został adaptowany z formatu COFF, tj. z formatu, który był używany po a.out, ale przed ELF;

- format PE można traktować jako pewną wersję formatu COFF opracowaną przez Microsoft; format PE przewidziany jest do stosowania w systemach 32-bitowych (Windows NT, Windows 95/98, . . .); implementacje Windows NT na różnych platformach (Pentium, Power PC i inne) używają tego samego formatu PE, aczkolwiek ze względu na inne kody binarne instrukcji nie jest możliwe uruchomienie programu wykonywalnego PE na innej platformie, niż ta dla której został utworzony;

- ogólnie, w systemie Windows rozwinięto pewne środowisko przeznaczone dla stosunkowo wolnych procesorów, ograniczonego rozmiaru pamięci RAM, a pierwotnie nawet bez stronicowania sprzętowego; stosowanie współdzielonych bibliotek pozwoliło na zaoszczędzenie sporo pamięci; wprowadzono też szereg trików, spośród których niektóre widoczne są w formacie PE/COFF;

- w formacie PE wyodrębniono zasoby programu (kursory, ikony, mapy bitowe, menu i fonty) — obiekty te współdzielone przez program i przez GUI;

- pliki w formacie wykonywalnym (ang. executable) PE są przewidziane dla środowiska ze stronicowaniem, tak że strony z PE są bezpośrednio odwzorowywane do pamięci i wykonywane podobnie jak strony formatu ELF;

- plik PE może występować zarówno w postaci programu EXE jak i biblioteki DLL (tj. biblioteki linkowanej dynamicznie) — w obu przypadkach format jest identyczny, a typ wskazywany jest przez jeden bit;

- każdy plik może zawierać listę funkcji eksportowanych i danych, które mogą być używane przez inne pliki PE ładowane do tej samej przestrzeni adresowej;

- w pliku PE zawarta jest również lista funkcji importowanych i danych — lista ta musi rozwiązana (ang. resolved) względem innych PE w czasie ładowania;

- każdy plik składa się z sekcji (nazywanych też segmentami, tak jak w formacie ELF);

- plik PE zaczyna się od małego programu DOSowego, który wyświetla napis "Program wymaga systemu Windows";

- pole na końcu nagłówka DOS EXE wskazuje na znaki PE, po których następuje nagłówek pliku (ang. file header) zawierający sekcję COFF i nagłówek opcjonalny, który pomimo nazwy pojawia się we wszystkich plikach PE i zawiera nagłówki sekcji (ang. list of section headers);

- poniżej podano strukturę nagłówka pliku w formacie PE;

```
char  signature[4] = "PE\0\0";   // identyfikator (magic number)
                                 // określa także porządek bajtów

// nagłówek formatu COFF

unsigned short  Machine;         // typ procesora:
                                 // 0x14C for 80386, itd.
unsigned short  NumberOfSections;       // liczba sekcji
unsigned long   TimeDateStamp;    // czas utworzenia lub zero
unsigned long   PointerToSymbolTable;       // offset tablicy
                                 // symboli (COFF) lub zero
unsigned long   NumberOfSymbols;       // liczba pozycji
                                 // w tablicy symboli
unsigned short  SizeOfOptionalHeader;       // rozmiar
                                 // nagłówka opcjonalnego
unsigned short  Characteristics;      // 02 = executable,
                                 // 0x200=nonrelocatable,
                                 //  0x2000 = DLL
```

Własności formatu PE

- format PE jest profesjonalnym formatem dla systemów operacyjnych z adresowaniem liniowym korzystającym z pamięci wirtualnej;

- dziedzictwo systemu DOS widoczne jest tylko w niewielkim stopniu;

- przed wprowadzeniem formatu PE używane były rozbudowane wersje formatu EXE, oznaczone symbolami NE i LE; format NE używany był dla 16-bitowych programów Windows, LE — dla 32-bitowych; w systemie OS/2 używany jest też format LX, natomiast format PE został wprowadzony wraz z wersją 3.1 systemu Windows NT;

- format PE posiada własności pozwalające na przyspieszenie ładowania programów w małych systemach — jednak ich skuteczność w współczesnych systemach 32-bitowych jest wątpliwa;

- format NE dla 16-bitowego kodu wykonywalnego był znacznie bardziej skomplikowany, tak że PE stanowi niewątpliwy postęp.

Struktury kodu i danych

- podział programu na moduły powoduje pewne rozproszenie kodu i danych – mimo tego należy dążyć by po integracji kod i dane programu zajmowały zwarte obszary pamięci;

- w przypadku procesorów o architekturze segmentowej oznacza to, że po integracji kod i dane programu zostaną umieszczone, odpowiednio, w segmencie kodu i w segmencie danych; należy przewidzieć też segment stosu potrzebny do przechowywania danych tworzonych dynamicznie;

- zatem segmenty kodu i segmenty danych (statycznych) z poszczególnych modułów trzeba połączyć razem, tworząc pojedynczy segment kodu i pojedynczy segment danych;

- w środowisku komputerów PC z procesorem Pentium przedstawione postulaty zrealizowano poprzez zdefiniowanie *segmentów programu*, stanowiących programowe odpowiedniki segmentów zdefiniowanych w architekturze procesorów Pentium;

- dyrektywa SEGMENT i związana z nią dyrektywa ENDS wskazują, odpowiednio, początek i koniec segmentu programu; parametry dyrektywy SEGMENT wpływają na przebieg linkowania a nie kompilacji; przykładowa postać tej dyrektywy może być następująca:

```
testy    SEGMENT  word  public  'rozkazy'
```

- parametry dyrektywy SEGMENT pozwalają określić:
  - własności adresu początkowego (np. że segment musi rozpoczynać się od adresu parzystego);
  - sposób łączenia segmentów o jednakowych nazwach;
  - pożądaną kolejność segmentów w programie po linkowaniu.

- segmenty programu są definiowane przez kompilatory języków wysokiego poziomu w postaci niejawnej, natomiast segmenty w postaci jawnej występują w programach asemblerowych; segmenty definiuje się za pomocą dyrektywy SEGMENT — poprzez określenie odpowiednich parametrów tej dyrektywy można uzyskać segmenty o pożądanych własnościach, co pozwoli na ich późniejszą integrację;

- kompilatory języków wysokiego poziomu tworzą zazwyczaj segmenty o podanych niżej nazwach

_TEXT — segment zawierający kod (rozkazy) programu;
_DATA — segment zawierający dane, o ustalonych wartościach początkowych, które program może zmieniać;
_BSS — segment zawierający dane niezainicjalizowane;
CONST — segment zawierający dane, o ustalonych wartościach początkowych, które nie mogą być zmieniane przez program;
STACK — segment stosu

- niekiedy zamiast pełnej integracji segmentów, tworzy się grupy segmentów o podobnych własnościach — segmenty w grupie zachowują swoją odrębność, ale dostęp do zawartych w nich danych realizowany jest tak, jak gdyby segmenty te stanowiły pojedynczy, zintegrowany segment; grupowanie segmentów realizuje się za pomocą dyrektywy GROUP;

- istotne ograniczenia występowały przy adresowaniu 16-bitowym, przy którym rozmiary segmentów nie mogą przekraczać 64 KB; z tego względu większe programy musiały być dzielone na segmenty o rozmiarach nie przekraczających 64 KB;

- podział programu na segmenty w pewnym stopniu komplikował dostęp do danych, wymagał używania dłuższych adresów (tj. zapisanych na większej liczbie bitów), a także utrudniał przekazywanie sterowania; w tej sytuacji starano się, by liczba segmentów w programie była możliwie jak najmniejsza, najlepiej by program zawierał jedynie segment kodu, segment danych statycznych i danych dynamicznych (segment stosu) — w takim przypadku zarówno adresy zmiennych jak i adresy instrukcji sterujących mogły być 16-bitowe;

- jednak, w trybie 16-bitowym, postulaty dotyczące budowy programu zawierającego wyłącznie segment kodu, segment danych statycznych i segment danych dynamicznych często nie mogły być spełnione, zwłaszcza w odniesieniu do większych programów.

- w celu uporządkowania możliwych sytuacji w zakresie adresowania w trybie 16- i 32-bitowym, programy wielosegmentowe podzielono na kilka klas, zależnie od sposobu adresowania danych i instrukcji; klasy te określano jako tzw. *modele pamięci* albo struktury programów; wyróżnia się 7 modeli pamięci:

tiny — program zawiera pojedynczy segment, w którym zawarte są kod i dane;
small — program zawiera pojedynczy segment kodu i pojedynczy segment danych;
medium — program zawiera pojedynczy segment danych i wiele segmentów kodu;
compact — program zawiera wiele segmentów danych i pojedynczy segment kodu;
large — program zawiera wiele segmentów danych i wiele segmentów kodu;
huge — odmiana modelu large: dopuszczalne jest definiowanie danych o wielkości przekraczającej 64KB;

flat — model używany w trybie 32-bitowym — program zawiera pojedynczy segment kodu i pojedynczy segment danych, przy czym rozmiary segmentów mogą dochodzić do 4 GB;

- dla każdego z tych modeli producenci oprogramowania dostarczali odrębny zestaw programów bibliotecznych (np. w systemie Borland C++ dostępne są biblioteki cs.lib dla modelu small, cm.lib dla modelu medium, itd.);

- w języku wysokiego poziomu wybór modelu pamięci dokonywany jest zazwyczaj poprzez podanie opcji kompilatora, natomiast w asemblerze używa się do tego celu dyrektywy .model